# Core Algorithm Overview

**State Problem:**

The purpose of this project is to create a heuristic delivery algorithm similar to the "Traveling Salesman Problem'' as well as creating a hash table data structure to store the packages from a csv file (WGUPSPackageFile.csv). The project will use a command line interface the interface will take as input a time in format HH:MM in military time, and the ID number of the package. The project will output screenshots of all packages at times: 9:00 AM, 10:30 AM, 12:03 AM, as well output the screenshot of the package queried by the user in the command line interface. The algorithm must complete all routes under 140 miles, and display the miles traveled by all trucks in that day. The project must all meet all deadlines, delays, and special note requirements. The project will import the mileage information of two connecting areas from a csv file (WGUPSDistanceTable.csv).

**Package Delivery Greedy Algorithm**

The algorithm used in this project is a greedy algorithm that makes decisions based on optimal choice at that point in time. The distance table is stored in a 2D list and the algorithm iterates through the list looking for the optimal mileage (less is better).
The algorithm in this project works as follows:

1. Iterate through the addresses in the distance list
2. Iterate through the packages in the truck inside the previous iteration.
3. Check for an address match
4. If there is a match, look at the current location index and compare distance, if the distance is less than the previous shortest distance, set the current distance as the minimum distance.
5. Once the both iterations are complete, add the mileage to a class variable.
6. Repeat this process for every package in the truck.
7. Once all packages have been delivered, add the mileage from the last location to the Hub.

**Pseudo code**

```
variables:
    int current_loaction
    Object Truck
    list [][] distance_table
    HashTable package_list
    int return_Hub = 0
    float min_distance
    int package_ID
    int index_column = 0

method:
while len(Truck.load) > 0:
```

The algorithm will loop as long as there are packages in the Truck Object. At the end of the algorithm the Truck.load is decreased by one when the package is delivered, and therefore the loop will end when all packages have been delivered.
The Truck Object keeps track of time, load list, miles, max size of load, speed of travel, and the truck number for each Truck defined in main.py.

```
|       for index_row = 0, index_row < len(distance_list), index_row += 1 :
|       |       for packages in Truck.load:
|       |       |       if distance_list[index_row][1] contains packages[1]:
```
This is used for finding an address match from the distance_list and the packages.
```
|       |       |       |       if distance_list[index_row][index_column + 2] != ‘’:
```
The “+2” in the index column is there because the first two indexes of the distance table are: the name of the location in that row, and the address of the location in that row. This means that the first two indexes do not contain any distance data. If the index column distance data is an empty string then the distance data must be looked at from the row of the column location.
```
|       |       |       |       |       current_distance = distance_list[index_row][index_column+2]
|       |       |       |       |       if current_distance < min_distance:
|       |       |       |       |       |       min_distance = current_distance
|       |       |       |       |       |       package_ID = packages[0]
|       |       |       |       |       |       current_location = index_row
```
If a package meets all the requirements for the shortest distance, changing these variables will keep track of the current location, the package ID of the package to be delivered, and the distance that will be traveled from the current location location to the new current location. This logic will be used again for the instances in which the distance_list[index_row][index_column + 2] is an empty string.
```
|       |       |       |       elif distance_list[index_column][index_row + 2] < min_distance:
```
Since the CSV is a is a two way table with mirroring values, flipping the index_column and index_row will arrive at the value that belongs where the empty string resides in distance_list[index_row][index_column+2]. The “+2” is there for the same reason it was used previously. The first two indexes per row are for the name of the location, and the address of the location respectively. The naming convention can be confusing since we did flip the indexing order, but the first ‘[]’ index corresponds to the row value of the distance table, and the second ‘[]’ will correspond to the column value of the distance table regardless of the indexing names.
```
|       |       |       |       |       min_distance = distance_list[index_column][index_row]
|       |       |       |       |       package_ID = packages[0]
|       |       |       |       |       current_location = index_row
```
These variables are used to keep track of the “greedy” minimum distance.
```
|       index_column = current_location
|       Truck.mileage(min_distance)
|       Truck.deliver(package_list, package_ID)
```
After The algorithm has iterated through all packages in the Truck.load the “greedy” solution will be the minimum distance from the current location to a new location. The algorithm will always choose the minimum distance from the current location. The index_column is an index of the current location from the truck's perspective. Setting the index_column to the current_location which was the optimal index_row, moves the new current location to that index value.
The Truck.mileage method keeps track of the sum of all miles traveled. The Truck.deliver method removes the package from Truck Objects internal list(Truck.load), and sets the delivery time of the package inside the package_list which is an instance of the user defined data structure that contains all the packages imported from the CSV(WGUPSPackageFile.csv).
```
Truck.mileage(distance_list[current_location][return_Hub + 2])
current_location = return_Hub
```
The final part of the algorithm is that it must return back to the Hub. Using the Truck.mileage method to get the mileage value from the last location to the hub. Lastly setting the current_location to return_Hub which is the index location of the hub(zero). This is to keep track of current_location at every point of the algorithm.

**Space-Time Complexity:**
The time complexity for the project is dependent on the greedy algorithm which is dependent on the size of the distance list. While there are 3 loops for this algorithm, the first and the third are dependent on the size of the load. The worst case for the size of the load is 16 for this project, the size of the load is an immutable variable in this case. The algorithm worst case would be 16 * N * 16, N being the size of the address list, therefore time complexity for this algorithm would be O(N). Within the algorithm the Truck method Truck.deliver is called to modify the list of all packages. However, since it is a hash data structure, modifying an object in that list using its ID takes O(1) thus not affecting the time complexity. Loading the packages to a hash data structure is also done manually using the ID of the package, taking O(1) also not affecting the time complexity. In the file TimeStamp.py there is a function called total_mileage()  which runs through all the packages in the hash table, the time complexity for that function is O(M), M being the size of the hash table. The time complexities for both are independent of each other, making the time complexity of the program O(N+M) simplifying to O(N). So as it stands the time complexity for the project is O(N).
The space complexity is dependent on the size of the distance list and the size of the package list. The package list grows at a linear rate O(M), however since the distance list is a two dimensional list it grows at a quadratic rate O( $N^2$ ). The space complexity would be O( $N^2$ +M), N being the dominant variable we can simplify it to an O( $N^2$ ) space complexity.

**Scalability and Maintainability**
The project's ability to scale with packages and addresses does not require much grunt work. Simply adding more packages and addresses will result in no issues. However, there is no current function for auto loading the packages, scaling in that aspect will mean having to create an auto load function.
The project's maintainability is similar to that of the scalability, there are some areas such as the user defined time class that will have to be migrated to a more industry standard library, this is also something that impacts the scalability of the project. The project is efficient in that there are functions or methods that are not used in the project. However using outside libraries is often required for standardization.

**Strengths**
1. The algorithm runs with time efficiency O(N).
2. The algorithm is simple and easy to maintain and adjust.

**Alternate Algorithms**
1. Dijkstra algorithm
    - The Dijkstra algorithm was originally used to find the shortest path between two nodes.
    - It can find a heuristically better optimized distance than the algorithm used in this project, in O(VE logV), V being vertices, E being the edges.
    - This algorithm uses a graph data structure, so converting the distance table to a graph would be required.
2. Depth First Search
    - The DFS algorithm used for traversing graphs or trees, it explores the node branch as far as possible before it backtracks.
    - It also can find a heuristically better optimized distance, because it would check alternate routes, instead of making a decision at each step. Its run time is O(|E| + |V|), E being the edges, V being the vertices.
    - This algorithm also uses a graph or tree data structure, so converting the distance table to either would be required to use.

## Self Adjusting Data Structure

**Hash Table**
The User defined data structure for this project was a hash table used to store the packages using their ID's with the hash function. The data structure uses a parameter to determine the size of the list used to create the hash table. The size of the list is also used as the hash key used for interesting and modifying data.

**Strengths and weaknesses**
The Strength of the self adjusting data structure used in this project is the efficiency of inserting and updating data in the hash table, it is a constant time operation as long as the ID of the data is used for the operation. However, since the size of the table is used as the hash key, the size of the hash table must be predefined. This could be an issue for data where the amount of data is unknown at the time of creating the hash table. Another potential limitation would be if the ID convention were to change, it would require an update to the code base.

**Overhead and Efficiency**
One of the great things about hash tables is the ability to look up objects in the list using the hash key, it is O(1) time efficient. This is because the hash key is used to reference the index of the list which is an operation of constant time. Adding more packages to the hash table will not affect this operation because the size of the list is used as the hash key, which avoids package collision.
The overhead of the data structure used in this project is O(N). With the growth of the amount of packages the hash table will also increase the size of the table.
The implications of this means that any changes to the amount of trucks or addresses will not impact the lookup function since it is reliant on the package ID. The package ID should be unique to each package thus not affecting the time or space complexity of the hash table used to store the packages.

**Alternate Data Structures**

1. Stack
    - A Stack is a collection of elements, only adding new objects to the top of the stack, and only removes objects from the top of the stack as well.
    - The benefits of a stak are that insertion and deletion are a constant time operation.
    - The downside to a stack is that an object look up function would require O(n).

2. Binary Search Tree
    - A Binary Search Tree is a hierarchical data structure with a root node, and 2 children nodes, used to represent linked nodes.
    - The lookup time for an object in a BST is O(h), h being the height of the tree.
    - One advantage of BST is the lookup time without an ID is still O(h), h being the height of the tree, for a balance tree being O(log2 N), N being the amount of items on the Tree. Considerably less than O(N), N being the size of the list, for the hash table.

**Different Approach**

Looking back at the project one of the things that I would do differently would be to implement the distance table from a 2D list to a graph. This would allow for the usage of more sophisticated algorithms such as the Dijkstra algorithm. I would also make changes to the user interface, allowing for key words such as “Truck”, “Screenshot” to be used to declutter the command line window at first run time.

**Development Environment**

**Hardware:**

**MacBook Pro**

| | |
|---|---|
| **Model Identifier:** | **MacBookPro13,1** |
| **Processor Name:** | **Dual-Core Intel Core i5** |
| **Processor Speed:** | **2 GHz** |
| **Number of Processors:** | **1** |
| **Total Number of Cores:** | **2** |
| **L2 Cache (per Core):** | **256 KB** |
| **L3 Cache:** | **4 MB** |
| **Hyper-Threading Technology:** | **Enabled** |
| **Memory:** | **8 GB** |
| **System Firmware Version:** | **429.60.3.0.0** |
| **IDE** | **PyCharm Community Edition 2020.3** |
| **Python** | **Python 3.9** |